Aplicación web para la gestión y búsqueda de eventos.

Universidad complutense de Madrid

Trabajo de Fin de Grado del Grado en Ingeniería Informática

Facultad de Informática

Autor:
Rafael Gómez Bermejo

Tutor:
Manuel Montenegro Montes

8 de junio de 2018

UNIVERSIDAD
COMPLUTENSE
MADRID

Índice general

Agradecimientos

Agradecer a Manuel Montenegro su dedicación y disposición en todo momento a ayudarme y aconsejarme a lo largo de todo el proyecto, haciéndolo posible.

Especial mención a Ana María Bermejo Ares por el diseño del logo empleado para la web.

Resumen

El objetivo principal de este trabajo de fin de grado es desarrollar un portal web que permita una búsqueda de eventos y actividades de ocio teniendo en cuenta determinados criterios tales como la proximidad geográfica y el coste del mismo, entre otros. Este proyecto nace para cubrir una necesidad de aquellos usuarios que deseen, en un determinado momento, asistir a alguna actividad de ocio (concierto, exposición, etc.) que se realice en un lugar cercano a aquel en el que ellos se encuentren.

Se pretende ofrecer una herramienta que ayude a un usuario a gestionar las actividades a realizar en su tiempo libre, aportando facilidades para su organización en función del perfil de cada usuario y sus capacidades. Para poder llevar a cabo este propósito se investigaron los factores más críticos para favorecer el aprovechamiento del tiempo y se desarrolló una herramienta web, haciendo uso de la tecnología *Django*, donde aquellas personas que busquen sacarle el máximo provecho a su tiempo libre acudan para gestionar todas sus actividades. En particular, la aplicación está enfocada a la búsqueda de actividades y eventos de ocio, poniendo énfasis en la posibilidad de asistir a dicha actividad en ese mismo momento. Para ello se tienen en cuenta factores como la cercanía física, la hora de comienzo, el precio de la misma, etc.

El resultado final ha cumplido satisfactoriamente con el objetivo buscado, pudiéndose lanzar al mercado competitivo. Además cuenta con un gran potencial de mejora para adaptarse a las necesidades futuras.

El código fuente de este proyecto se encuentra disponible en la siguiente dirección: `https://github.com/RafaelGB/Eventies`

Palabras clave

Gestión de eventos, framework web, Django, recomendación de eventos, filtrado de eventos, acceso a base de datos, geoposicionamiento.

Abstract

The aim of this final degree project is to develop a web site that allows one to search for leisure activities and events by taking different criteria into account, such as geographical proximity, and the price of attending the event. This project was conceived as a way to fulfil the needs of those users who decide, at a given moment, to attend to a leisure activity (concert, exhibition, etc.) that takes place nearby.

It is intended to offer a tool that helps a user to manage the activities to be carried out in their free time, providing facilities for their organization according to the profile of each user and their capabilities. In order to carry out this purpose, the most critical factors were investigated to favor the use of time, and a web tool was developed, making use of the technology *Django*, where those people looking to make the most of their free time go to manage all their activities. In particular, the application is focused on the pursuit of leisure activities and events, emphasizing the possibility of attending this activity at that moment. For this, factors such as physical proximity, start time, price, etc. are taken into account.

The final result has fulfilled satisfactorily the goal sought, being able to be launched in the competitive market. It also has a great potential for improvement to adapt to future needs.

The source code of this project is available at the following address: `https://github.com/RafaelGB/Eventies`

Keywords

Event management, web framework, Django, event recommendation, event filtering, database access, geopositioning.

Capítulo 1

Introducción

El objetivo de este trabajo de fin de grado es el desarrollo de una aplicación de web social centrada en eventos y actividades de ocio (conciertos, películas, exposiciones, etc.).

En este capítulo se expondrá la motivación de llevar a cabo una página web para la gestión y búsqueda de eventos así como los objetivos que se pretenden lograr con su implementación. Para concluir se explicará el plan de trabajo que se ha seguido, comentando qué objetivo se debe cumplir en cada una de las fases del desarrollo.

1.1. Motivación

Con una sociedad de gustos cada vez más heterogéneos, existen numerosos tipos de actividades que cubren todo tipo de motivaciones, siendo estas las que nos definen como personas.

Por toda la red hay disponibles un sinfín de herramientas para descubrir el evento que mejor concuerde con las preferencias de un usuario, pero este proyecto trata de acercar de una manera sencilla su creación y organización, además de las funcionalidades que ya se ofrecen en el resto de aplicaciones.

Dado que el mundo del ocio es un mercado en constante evolución, una aplicación que pretenda gestionarlo necesitará adaptarse rápidamente sin un excesivo tiempo de desarrollo, por lo que otro objetivo de este proyecto es crear una estructura suficientemente modularizable como para seguir este exigente ritmo evolutivo.

1.2. Objetivos

A continuación se enumerarán y describirán los objetivos del TFG:

- **Implementación de la creación de cuentas de usuario y su gestión**

  Implementar la funcionalidad de crear una cuenta, recuperar la contraseña de una cuenta y gestionar los datos respecto a dicha cuenta (nombre de usuario, email, avatar, etc.) usando las herramientas de que ofrece *Django* para ofrecer una seguridad razonable.

- **Implementación de la gestión de los eventos**

  Implementar la funcionalidad de crear y editar eventos para usuarios registrados en la aplicación y la funcionalidad de mostrar el contenido de un evento para cualquier usuario.

- **Implementación de un buscador y herramientas de filtrado**

  Desarrollar una interfaz donde poder buscar los eventos deseados en una lista y aplicar filtros para facilitar dicha tarea.

- **Integrar un sistema de recomendaciones**

  Crear un sistema de recomendaciones basado en las interacciones del usuario con la página web y sus gustos.

1.3. Plan de trabajo

En esta sección se detalla la organización seguida para el desarrollo del proyecto.

1.3.1. Estudio de las herramientas y tecnologías

Durante el primer período de las fases de desarrollo se investigarán las diferentes opciones para llevar a cabo una página web. La primera decisión será elegir un *framework* sobre el que cimentar toda la implementación, ya que esta elección condiciona el resto de fases. Se contemplará la decisión de usar *Django* frente a otros frameworks como *Spring*, sobre el que el autor tenía ya experiencia, para poder aprender otras tecnologías y lenguajes, ya que *Django* está desarrollado en *Python*, un lenguaje cada vez más utilizado por su versatilidad y sobre el cual no había cursado ninguna asignatura durante mis estudios en el grado. También se utilizará *Django* ya que el proyecto se concibió inicialmente como un sistema recomendador

de eventos, lo cual justificaría el uso de *Python* junto con las librerías de *machine learning*, como `Sklearn`.

Para llevar a cabo la interfaz web se estudiará el uso de la librería de *Bootstrap* debido a su gran presencia en el mercado de desarrollo de páginas web.

Como entorno de desarrollo elegiré *Sublime text* tras probar diferentes opciones.

En el capítulo 3 se describen con más detalle cada una de las tecnologías usadas durante el proyecto.

1.3.2. Proceso de desarrollo para cada funcionalidad

Primero se diseñará una base sobre la cual enlazar las diferentes páginas web. Una vez se identifican los elementos en común que definirán la interfaz, se pasa a una segunda fase donde se trabajaba cada plantilla HTML por separado, terminando por pulir los fallos. Este proceso se repetirá con cada característica básica de la web.

En paralelo se desarrolla la parte del lado del servidor haciendo uso de *Django* para tratar la información recibida. De este modo se define cómo deben relacionarse las diferentes plantillas entre sí. Al ser necesario guardar la información recogida y que esta perdure en el tiempo, se utilizará una base de datos para almacenar y gestionar la información.

1.3.3. Implementación de la creación de cuentas y su gestión

El desarrollo partirá de la implementación de las cuentas de usuario donde poder autenticarse en la aplicación. Para ello su usará el sistema de autenticación ofrecido por *Django* y se ampliará con las características extra que requieran los usuarios de la web. Para que el usuario, en caso de olvido de su contraseña, pude recuperarla mediante su correo electrónico, se usará un *middleware* que escuche los correos enviados y se muestren por terminal. De este modo se podrá probar su funcionamiento.

1.3.4. Implementación de la gestión de los eventos

Una vez se dispone de una estructura con usuarios, pasaré a implementar la creación y edición de los eventos a través de formularios (con la condición de estar registrados) y la interacción con la base de datos.

Para que otros usuarios puedan acceder a la información de los eventos creados por

los demás, pasaré a añadir una interfaz donde el usuario podrá acceder a la información de dichos eventos e interactuar con ellos añadiendo comentarios e indicando si asistirá al evento, le gusta o no le gusta.

1.3.5. Implementación de un buscador y herramientas de filtrado

Tras tener una base donde se gestionan los eventos pasaré a desarrollar un sistema de búsqueda de eventos que despliegue los resultados de búsqueda en una lista y me permita filtrar por diversos criterios.

1.3.6. Integración de un sistema de recomendaciones

Tras pulir todas las características principales de la web se dispondrá de una base sobre la que usar la información recogida para informar al usuario sobre otros eventos que podían interesarle. Tras descartar por falta de tiempo y coherencia el uso de las librerías de *machine learning*, implementé un algoritmo donde se tenía en cuenta los eventos a los que el usuario había asistido y buscaba entre los demás asistentes otros eventos afines.

Capítulo 2

Introduction

The objective of this final degree project is the development of a social web application oriented to events and leisure activities (concerts, films, exhibitions, etc.).

In this chapter we will describe the motivation to carry out a web page for the management and search of events as well as the objectives that are intended to be achieved with its implementation. To conclude, the work plan that has been followed will be explained, highlighting the objectives to be met in each of the phases of development.

2.1. Motivation

In a society of increasingly heterogeneous tastes, there are several types of activities that cover all kinds of motivations, these being what define us as people.

All over the internet there is an endless number of tools available to discover the event that best matches the preferences of a user, but this project tries to bring its creation and organization in a simple way, in addition to the functionalities already offered by the rest of applications.

Given that the world of entertainment is a market in constant evolution, an application that intends to manage leisure activities will need to adapt quickly without excessive development time, so another objective of this project is to create a sufficiently modular structure to follow this demanding evolutionary rhythm.

2.2. Objectives

The objectives of the TFG will be listed and described below:

- **Implementation of account creation and management**

  Implement the functionality of creating an account, recovering the password of an account and managing the data regarding that account (username, email, avatar, etc.) using the tools offered by *Django* to offer a sensible level of security.

- **Implementation of event management**

  Implement the functionality of creating and editing events for users registered in the application and the functionality of displaying the content of an event for any user.

- **Implementation of a search engine and filtering tools**

  Develop an interface where one can search the desired events in a list and apply filters to facilitate this task.

- **Integrate a system of recommendations**

  Create a recommendation system based on the user's interactions with the website and their tastes.

2.3. Workplan

In this section is detailed the organization followed for the development of the project.

2.3.1. Study of tools and technologies

During the first period of the development phases, we will survey the different options to carry out the development of web page. The first decision will be to choose a *framework* on which to base the entire implementation, since this choice determines the rest of the phases. We will contemplate the decision to use *Django* instead of other frameworks like *Spring*, (on which I already had experience) to be able to learn other technologies and languages, since *Django* is developed in *Python*, a language increasingly used for its versatility and which i had not been taught in my degree studies so far. I will also make the decision to use *Django* due to the original idea of giving more weight to the recommendation system and make use of the *machine learning* libraries that *Python* offers as `Sklearn`.

To develop the web interface, I will opt for the *Bootstrap* library due to its great presence in the web development market.

As a development environment I will choose *Sublime text* after trying different options.

2.3.2. Development process for each functionality

First, a base will be designed on which to link the different web pages. Once the common elements that will define the interface are found, a second phase will start, where each HTML template was worked on separately, finishing by polishing the errors. This process will be repeated for each basic feature of the web.

In parallel, the server-side part will be developed using *Django* to process the received information, thus defining how the different templates should relate to each other. As it is necessary to keep the information collected so it persist over time, each information treatment will be integrated into the database.

2.3.3. Implementation of account creation and management

The development will start with the implementation of user accounts where one can authenticate in the application. In order to do this, the user will use the authentication system offered by *Django*, which will be expanded with the extra features required by web users. In case the user has forgotten their password, the system will allow them to recover it via e-mail. A *middleware* will be used to capture the event of an email being sent, in order to log this event.

2.3.4. Implementation of event management

Once the functionality involving user management is available, I will start to implement the creation and edition of the events through forms, which will be available provided the user has been registered before.

In order that other users can access the information of the events created by others, I will add an interface where one can access the information of these events and interact with them by adding comments and by specifying whether he is going to attend the event, and whether he is interested in it or not.

2.3.5. Implementation of a search engine and filtering tools

After having a base where the events are managed, I will continue by developing an event search system that displays the results in a list and allows the user to filter them according to several criteria.

2.3.6. Integration of a system of recommendations

After polishing all the main features of the web there will be a basis on which to use the information collected in order to inform the user about other events that could interest him. After discarding the use of *machine learning* libraries due to lack of time and coherence, I implemented an algorithm that took the events that the user had attended into account, and searched for other related events from the other attendees.

Capítulo 3

Selección de herramientas y tecnologías

En este capítulo se detallarán tanto las herramientas usadas en el proyecto como las descartadas, su función y cómo se han empleado o se pretendía emplear.

3.1. Herramientas y tecnologías usadas

3.1.1. Python [5]

Python es un lenguaje de programación de alto nivel orientado a objetos, dinámicamente tipado, con una sintáxis indentada, es decir, basada en la tabulación. Es un lenguaje diseñado para optimizar la productividad y disminuir el tiempo requerido para su mantenimiento. Su desarrollo está bajo una licencia *open-source* con un gran número de librerías tanto propias como *third-party* muy fáciles de integrar.

Se ha utilizado el lenguaje para toda la funcionalidad del lado del servidor, denominada *back-end*, que ofrece el framework Django del que hablaré más adelante y para hacer uso de sus librerías de *machine learning*, teniendo todo el tratamiento interno de la información bajo la misma estructura. Se ha usado la versión 3.6.3, siendo esta la más actualizada.

3.1.2. Django [3, 2]

Django es un framework Web basado en Python que abstrae al programador de muchas de las complicaciones del desarrollo de la página, fundamentalmente su estructura y su seguridad, de tal manera que se puede escribir la aplicación sin tener que reinventar la rueda. Es un framework *full-stack*, es decir, gestiona tanto la capa de presentación como la capa de acceso a datos.

Se diferencia de otros *frameworks* en que aprovecha la versatilidad de Python, que permite modularizar el código eficientemente permitiendo trabajar cada fragmento como una aplicación individual y sobre todo dando la capacidad de integrar fácilmente herramientas desarrolladas para Django y ahondando más en el concepto de la reutilización. Su modularización permite una escalabilidad potencial, ideal para unos requisitos en constante evolución como suponía este proyecto que partía de una idea genérica.

Django es un framework de alto nivel que impone una estructura determinada en los ficheros de conforman el proyecto. En la figura 3.1 se muestra la estructura de ficheros y los componentes de un proyecto estándar en Django. A continuación se pasará a explicar la función de cada elemento de la estructura:

- **Nombre_del_proyecto:** Django crea una carpeta cuyo nombre coincide con el proyecto donde se almacenan los ficheros de configuración del mismo.
  - `urls.py`: Fichero encargado de configurar el **mapa de rutas**[1] de la aplicación web. No se podrá acceder a ninguna ruta que no esté previamente definida aquí.
  - `settings.py`: Archivo encargado de las opciones generales de la aplicación. Las más relevantes a mencionar son la configuración de la base de datos y las aplicaciones instaladas, ya sean propias o de terceros usando librerías.
- **APPS:** Django permite modularizar una aplicación web en distintas subaplicaciones, cada una de ellas implementando la funcionalidad del lado del servidor mediante unas serie de modelos, vistas y de sus propias opciones de administración.
  - `views.py`: Base del controlador de Django que determina la vista a mostrar, y con qué valores se va a representar dicha vista.
  - `forms.py`: Archivo opcional, pero recomendable, que separa la lógica general del controlador de la lógica de los formularios, con funciones tales como la validación de los mismos o su inicialización.

[1] Conjunto de URLs válidas definidas para la aplicación web utilizadas para comprobar y redirigir cada solicitud proveniente del navegador a partir de su URL.

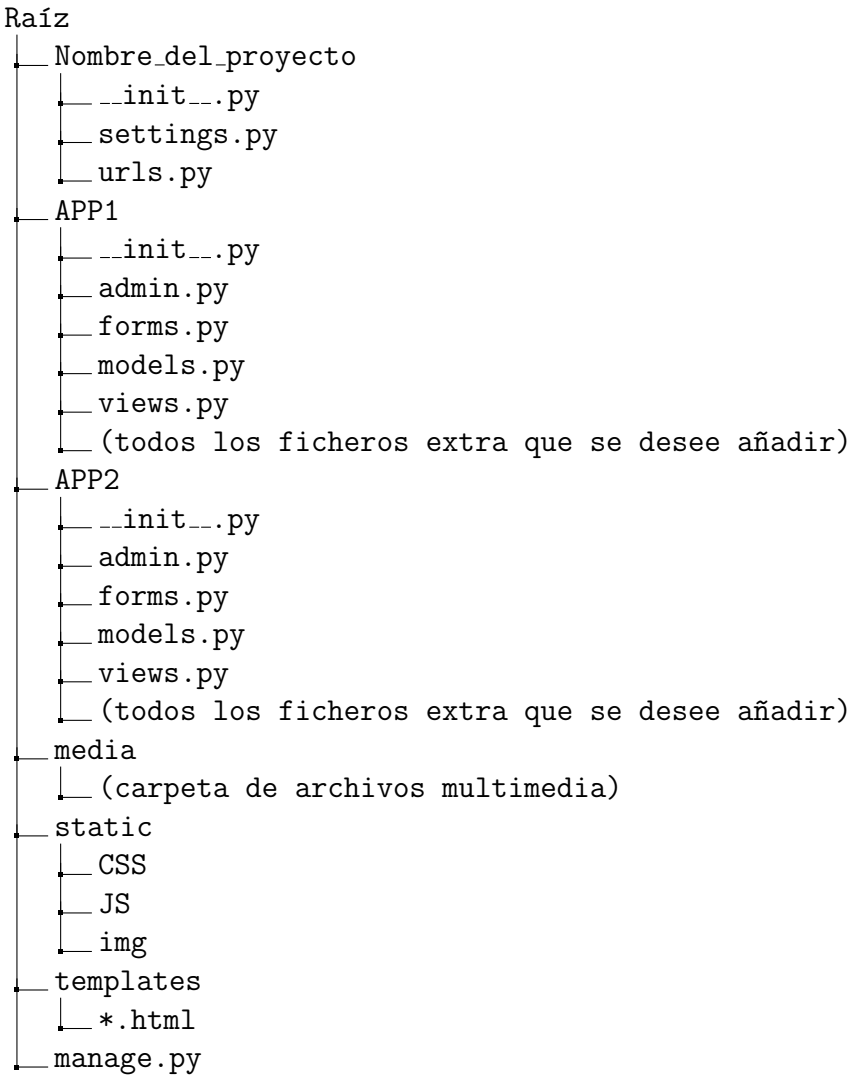

Figura 3.1: Estructura de ficheros de un proyecto Django.

    - **models.py:** Contiene toda la lógica de modelo de la aplicación, donde se hace uso del ORM integrado en Django para transformar los objetos Python, incluyendo sus relaciones con otros objetos, en filas de la base de datos y viceversa incluyendo todas sus posibles relaciones.
    - **admin.py:** Sirve para hacer uso de la funcionalidad de la administración por defecto que proporciona Django para los modelos creados en la base de datos, ofreciendo una interfaz gráfica que interactúa directamente con esta.

- **media:** Contiene el contenido multimedia dinámicamente generado tales como las fotos subidas por un usuario.
- **static:** Contiene todo el contenido estático usado en la vista (hojas de estilo, Javascript, imágenes).
- **templates:** Almacena plantillas, que son documentos HTML con marcadores. Las vistas rellenan estos marcadores con valores concretos.
- **manage.py:** Script principal de Django encargado de ejecutar las acciones principales por defecto y las incluidas en las aplicaciones y librerías añadidas (por ejemplo arrancar el servidor o construir la base de datos).

Para este proyecto se ha usado la versión 1.11.6.

3.1.3. Django ORM [1]

Django ORM (Object Relational Mapper) es la herramienta que proporciona Django para interactuar con una base de datos. Creando clases en Python basadas en la librería de su ORM se crearán sus correspondientes tablas en función de los atributos especificados en un objeto y su tipo. Podremos interactuar con esta tabla a través de *querysets*, que son una abstracción del concepto de consulta que permite leer los datos de una base de datos, filtrarlos y ordenarlos.

En todas las apps del proyecto se ha creado un fichero **models.py** donde se hace uso de esta herramienta.

3.1.4. Bootstrap [6]

Bootstrap es un framework dirigido al *front-end* de una aplicación web, por lo que está orientado a la parte que se ejecuta en el cliente, en este caso el navegador web. Utiliza HTML5, CSS y Javascript, los lenguajes por excelencia para contenido web, para hacer uso

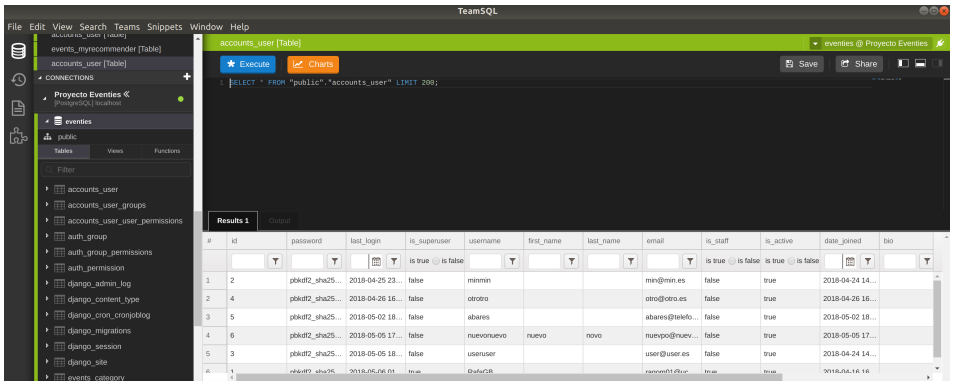

Figura 3.2: Captura de pantalla de TeamSQL

de sus librerías que permiten diseñar la web de una manera profesional creando al mismo tiempo un diseño adaptado para dispositivos móviles.

He elegido esta librería ya que se adapta perfectamente a las necesidades de listas y formularios del proyecto. Principalmente hago uso de la versión 3.0, ya que la última versión (la 4) se encuentra en fase de pruebas (beta). Pese a ello, esta última versión se encuentra en una fase de desarrollo avanzada, por lo que se ha utilizado en algunas partes de la aplicación, como la identificación de usuarios, su creación, y la recuperación de la contraseña de una cuenta.

3.1.5. PostgreSQL [8]

PostgreSQL es un sistema gestor de base de datos relacionales *open-source* con una comunidad activa y en constante desarrollo que permite crear tipos de datos definidos por el programador, siendo ideal para datos geométricos que no están soportados en *SQLite*, la base de datos utilizada por defecto en Django. En el proyecto se hace uso de la librería GeoDjango, por lo que el gestor de la base de datos a utilizar requería de un soporte de datos geométricos rápido y compatible con el framework.

Como editor de apoyo para depurar se ha usado TeamSQL, una interfaz gráfica para PostgreSQL compatible con Ubuntu (Figura 3.2).

3.1.6. Google Maps API [7]

Google Maps API es una popular herramienta de Google que permite mostrar al usuario un mapa de cualquier rincón del mundo con una amplia lista de opciones disponibles.

La web hace uso de esta API para mostrar la ubicación de cada evento. Además si el usuario permite el uso de la localización GPS en su dispositivo, muestra el camino más corto desde donde se encuentre en ese momento.

3.1.7. Sublime Text

Sublime Text es un completo editor de texto (inicialmente concebido como plug-in de Vim) que soporta Python, entre otros muchos lenguajes. Se ha elegido este editor por sus funciones de multi-layout que permiten visualizar hasta cuatro ventanas independientes. Su resaltado de paréntesis e indentación permiten la detección temprana de errores en la estructura del código fuente en Python. Su búsqueda dinámica, que admite expresiones regulares, ayuda a encontrar rápidamente los elementos deseados en todo el proyecto.

3.1.8. Git/Github

Github es una interfaz web que proporciona un sistema de control de versiones basado en Git. Facilita la organización de un proyecto colaborativo permitiendo visualizar los cambios efectuados a nivel de línea de código, así como algunas estadísticas del proyecto, tales como número de cambios y las fechas de estos.

Pese a tratarse de un proyecto individual, he elegido usar esta herramienta a modo de copia de seguridad del código fuente ayudándome así también a una mejor organización y seguridad a la hora de programar.

3.1.9. LaTeX[4]

LaTeX sistema de composición de textos basado en comandos utilizado TeX para obtener una tipografía de alta calidad e incluir expresiones científicas.

Se ha utilizado LaTeX para generar esta memoria. Para ello he hecho uso del editor de texto online Overleaf (Figura 3.3), el cual facilita la organización del código que genera el PDF y ofrece una vista previa en tiempo real que permite acceder al código rápidamente haciendo clic en el texto deseado del PDF.

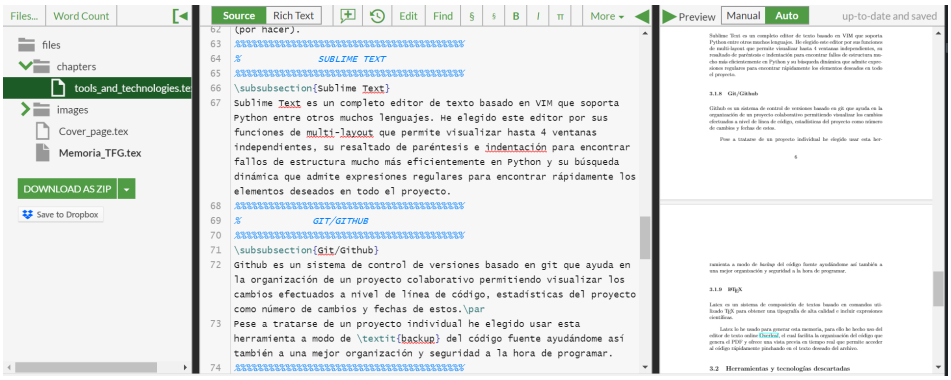

Figura 3.3: Captura de pantalla de Overleaf

3.2. Herramientas y tecnologías descartadas

3.2.1. AngularJS

AngularJS es un framework orientado a *front-end* para Javascript de documentos estáticos que hacen un uso amplio de AJAX, es decir, que no requieren de recargar la página para el envío y respuesta de información.

Pese a ser una herramienta cada vez más popular y completa, esta herramienta sería infrautilizada para este proyecto, ya que he optado por el modelo clásico de aplicaciones web, basado en saltos entre páginas para procesar los formularios, con un uso minoritario de AJAX.

3.2.2. Spring

Spring es un framework *full-stack* basado en Java para el tratamiento de datos, y JSP para implementar la parte visual. Siendo Java el lenguaje más popular a día de hoy, Spring cuenta con un gran soporte y mejora continuo teniendo herramientas especializadas para los patrones recurrentes utilizados en una aplicación web (configuración de seguridad, estructura de aplicación, etc).

Aun teniendo experiencia previa con Spring, decidí no basar el proyecto en este framework para aprender otras herramientas y lenguajes competitivos en el mercado. Otra razón de peso ha sido por la idea original de hacer un uso amplio de librerías de *machine learning*, que cuentan con un mejor soporte en Python.

3.2.3. MongoDB

MongoDB es el sistema de bases de datos NoSQL más popular. Guarda la información en ficheros BSON (codificación binaria de ficheros JSON) en lugar de tablas como una base de datos SQL relacional. Este tipo de bases de datos son ideales para el uso de técnicas *big data* y análisis de la información que aprovechan su estructura y rapidez de acceso.

Esta opción de almacenamiento fue descartada ya en fase de desarrollo al carecer de suficientes datos analizables que fueran aprovechables por las librerías de *machine learning* y sustituyéndose por la creación dinámica de ficheros CSV (ficheros de texto en los que los valores están separados por comas) donde se tratará la información. Además, el tipo de información que se maneja en el proyecto se adecua mejor a un modelo relacional basado en tablas que a uno basado en documentos.

Capítulo 4

Arquitectura del sistema

En este capítulo se profundizará en el funcionamiento de las herramientas desde un enfoque técnico y su integración en el proyecto.

4.1. MVC de Django

Django sigue el esquema de MVC (modelo, vista, controlador), pero con una interpretación propia distinta al modelo MVC clásico. A continuación se pasará a explicar en qué consiste este esquema y qué matices añade Django al MVC.

El patrón MVC separa en tres claros conjuntos la arquitectura de un programa:

- **Modelo:** Es la representación de los datos usados en el programa junto con la interfaz que permite su acceso desde los otros componentes de la aplicación. Se usa el modelo para comunicar la base de datos con el programa abstrayendo al desarrollador de las complejidades del tratamiento de la información que requiere la base de datos subyacente, por lo que se puede tratar independientemente el modelo de la base de datos, incluso trabajar con bases de datos diferentes simultáneamente.

- **Vista:** Es conocida como la capa de presentación que expone al modelo. La vista engloba todo aquello que puede ver el usuario final de la aplicación e interactúa con este.

- **Controlador:** Sirve de capa intermedia que comunica el modelo con la vista y viceversa. El controlador determina qué información se obtiene de la base de la base de datos para actualizar el modelo y qué información pasa a la vista.

Pese a que su arquitectura se asemeja a este modelo en capas, *Django* mantiene claras diferencias en su implementación, ya la parte del controlador queda definida por el propio Django, dejando al desarrollador un esquema de modelos, vistas y plantillas (MVP). A continuación se explica en más detalle dicho esquema:

- **Modelo:** es la capa de acceso a los datos y todo lo que concierne a estos (su lógica de acceso, su validación, su comportamiento y las relaciones entre ellos).

  El siguiente fragmento de código muestra la definición de un modelo y qué contiene:

```
from django.db import models

class MyModel(models.Model): # La clase hereda de models.Model
   # Se debe definir el tipo de dato y sus atributos
    texto = models.CharField(max_length=30)
    fecha = models.DateTimeField(default=datetime.now, blank=True)
    # Define la representacion del modelo como cadena de texto.
    def __str__(self):
        return self.texto
    # Metodos que permiten acceder a los componentes de un modelo
    @classmethod
    def metodo(self, entrada):
      # Tratar los parametros
        return resultado # opcional
```

- **Vista:** en esta capa se encuentra la lógica que accede a los modelos e invoca a las plantillas. Cuando se llama a una ruta en el navegador, la vista asociada es la encargada de llamar a la plantilla cargando los datos pertinentes.

  En el siguiente ejemplo se detalla la configuración de una vista de tipo `ListView` implementada por Django que permite cargar y mostrar una lista de un modelo:

```
from django.views.generic import ListView
from .models import MyModel

class Mi_vista(ListView):
    model = MyModel
    context_object_name = 'objetos_del_modelo'
    template_name = 'mi_plantilla.html'
    """
    ------------------------------------------------
        Funciones de la clase
    ------------------------------------------------
    """
    def get_context_data(self, **kwargs):
```

```
        context = super(Mi_vista, self).get_context_data(**kwargs)
    # Se puede incorporar todos los datos que se deseen al contexto
        return context
```

- **Plantilla:** es la capa de presentación. La plantilla es la encargada de determinar los elementos que deben aparecer en la página e interactúan con el usuario. Para su desarrollo se usan los lenguajes web HTML5, hojas de estilo y Javascript.

  A continuación se muestra un ejemplo de plantilla que dispone de una lista sobre un modelo cargado por la vista:

```
{% extends 'base.html' %} <!-- Las plantillas pueden
   heredarse -->

{% load propiedades1 %}
{% load propiedades2 %}

{% block stylesheet %}<!-- opcion de estilo -->
<link rel="stylesheet" href="{% propiedades1 'ruta de hoja
   de estilo' %}">
{% endblock %}
{% block content %}
    <div>
        <h1>Titulo</h1>
    </div>

    {% for objeto in objetos_del_modelo %}
        <div>
            <p>{{ objeto.texto }}</p>
            <p>{{ objeto.fecha }}</p>
        </div>
    {% endfor %}
{% endblock %}
```

Las diferencias más destacables de la interpretación de Django del MVC clásico se resumen en que la vista de Django en este caso complementa las funciones del controlador del MVC clásico y las plantillas hacen la función de la vista en el MVC clásico al tratarse de páginas web con marcadores.

4.2. Django ORM

En esta sección se detallará el funcionamiento del ORM (Object Relational Mapper) de Django presentado en el capítulo anterior.

Todos los modelos de Django son subclases de una clase preparada para proporcionar una API de acceso a la base de datos generada automáticamente.

Si como ejemplo se toma el siguiente modelo:

```
from django.db import models
"""
En todas las clases que implementan 'models' se inserta
automaticamente una clave primaria de la forma

id = models.AutoField(primary_key=True)

si no se especifica ninguna por defecto
"""
class Chef(models.Model):
    nombre = models.CharField(max_length=50)
    apellidos = models.CharField(max_length=50)

    def __str__(self):
            return self.nombre

class Receta(models.Model):
    creador = models.ForeignKey(Chef, on_delete=models.CASCADE)
    nombre = models.CharField(max_length=100)
    duracion = models.DurationField()
    calificacion = models.IntegerField()

    def __str__(self):
            return self.nombre
```

Django ORM proporciona opciones de creación, modificación, borrado y consulta.

En el siguiente ejemplo se crean objetos de los modelos definidos anteriormente:

```
"""
Se pueden incluir todos los valores dentro de la llamada a la funcion 'create' o
    indicarse por separado antes de guardarse
"""
# Se crea el objeto indicando los valores en la funcion
```

```
Peter = Chef.objects.create(nombre="Peter",apellidos="Garcia")
Peter.save()
# Se crea el objeto indicando los valores por separado
Marta = Chef.objects.create(nombre="Marta")
Marta.apellidos = "Lopez"
Marta.save()
# Una vez guardado el objeto dispondra de una clave primaria
clave_primaria = Marta.pk
# Se llama a la clave externa usando la variable asociada al modelo
Receta.objects.create(creador=Peter,nombre="pure de patatas")
# Se llama a la clave externa adquiriendo primero el modelo
getChef_marta = Chef.objects.get(pk=clave_primaria)
Receta.objects.create(creador=getChef_marta,nombre="arroz a la cubana")
```

Django ORM dispone de un sistema de consultas y filtros para acceder a la información de los modelos. A continuación se detallan breves ejemplos que explican su funcionamiento:

```
todo = Chef.objects.all() # devuelve toda la informacion almacenada en el modelo
#<QuerySet [<Chef: Peter>,<Chef: Marta>]>
filter = Chef.objects.filter(nombre="Peter") # Filtra por parametro-valor
#<QuerySet [<Chef: Peter>]>

for object in todo:
   # Se puede iterar sobre una consulta
```

Para modificar un dato en el modelo se guarda la nueva información sustituyendo el valor y guardando el objeto de nuevo, como se ve en el siguiente ejemplo:

```
Marta = Chef.objects.get(nombre="Marta")
Marta.apellidos = "Rodriguez"
Marta.save()
```

Para eliminar un objeto del modelo se hace uso de la función `delete()`:

```
Peter = Chef.objects.get(nombre="Peter")
Peter.delete()
# Si se consultan todos los elementos se observa que ha sido borrado
Chef.objects.all()
#<QuerySet [<Chef: Marta>]>
# Del mismo modo borra en cascada las recetas cuyo chef se ha eliminado
Receta.objects.all()
#<QuerySet [<Receta: arroz a la cubana>]>
```

4.3. Gestión de los usuarios

Django proporciona una herramienta integrada para la gestión de los usuarios, donde se incluye un modelo de usuario, un modelo de grupos de usuario, formularios incluyendo su validación y un sistema de autenticación.

Para cada grupo de usuarios se pueden delimitar los permisos que uno desee, dando o quitando acceso a las partes de la aplicación que el desarrollador crea oportunas.

Por defecto un usuario contará con los siguientes campos:

- `username`
- `password`
- `email`
- `first_name`
- `last_name`

No obstante, este modelo predefinido permite la posibilidad de añadir campos nuevos si la aplicación lo requiere. Para ello es necesario la creación de un modelo nuevo que implemente la clase `django.contrib.auth.models.AbstractUser`.

A continuación se muestra un ejemplo de gestión de usuarios con el modelo por defecto:

```
from django.contrib.auth.models import User
usuario = User.objects.create_user('Rafa', 'nombre@email.com', 'password')

# Pese a estar guardado ya en la base de datos
# es posible seguir integrando/cambiando valores
usuario.last_name = 'Gomez Bermejo'
usuario.save()
#----------------------------------------
from django.contrib.auth import authenticate
usuario = authenticate(username='Rafa', password='password')
if usuario is not None:
    # Código para un usuario autenticado
else:
    # Código para un usuario sin autenticar
```

Para el ejemplo de gestión de usuarios con un modelo personalizado se mostrará el modelo usado en el proyecto, añadiendo campos como una foto de perfil:

```
from django.db import models
from django.contrib.auth.models import AbstractUser

def get_image_filename(instance, filename):
    primaryKey = instance.pk
    return "Users/%s/%s" % (str(primaryKey), filename)

class User(AbstractUser):
    bio = models.TextField(max_length=500, blank=True,default="")
    location = models.CharField(max_length=30, blank=True,default="")
    birth_date = models.DateField(null=True, blank=True)
    avatar = models.ImageField(
           upload_to =get_image_filename,
           default = 'none/icon_user.png',
           )
    def __str__(self):
        return self.username
```

4.4. Gestión de los eventos

La aplicación de los eventos (*events*) en el proyecto supone el grueso del desarrollo del lado del servidor. El modelo principal que contiene toda la información básica es *Event*.

```
class Event(models.Model):
    """
              Informacion basica del evento
    ------------------------------------------------------------
    """
    title = models.CharField(max_length=30)
    description = models.TextField(max_length=5000)
    summary = models.CharField(max_length=50)
    # precio estimado
    budget = models.DecimalField(null=True, max_digits=5, decimal_places=2,
        validators=[MinValueValidator(Decimal('0.00'))], default=Decimal(0.00))
    # duracion estimada
    duration = models.DurationField()
    """
              Contadores creados por defecto
    ------------------------------------------------------------
    """
    # numero de visitas recibidas
```

```
views = models.PositiveIntegerField(default=0)
"""
                Fechas
-------------------------------------------------------

"""
# fecha en la que se celebraria el evento
date = models.DateTimeField(default=datetime.now, blank=True)
# fecha en la que el organizador crea el evento
created_at = models.DateTimeField(auto_now_add=True)
# fecha de la ultima modificacion llevada a cabo
updated_at = models.DateTimeField(null=True)
"""
                Relaciones uno a muchos
-------------------------------------------------------

"""
# define quien crea el evento
created_by = models.ForeignKey(User, related_name='events')
"""
                Relaciones uno a uno
-------------------------------------------------------
"""
# define la ubicacion donde se llevara a cabo el evento.
# es tratada a parte debido al uso de un tipo modelo que admite calculos
   geometricos
geopos_at = models.OneToOneField(
    Geolocation,
    on_delete=models.CASCADE,
    null=True,
)
"""
                Relaciones muchos a muchos
-------------------------------------------------------

"""
# lista de usuarios que estan interesados
interested_in = models.ManyToManyField(User,related_name='users_interested')
# lista de usuarios que no estan interesados
not_interested_in =
    models.ManyToManyField(User,related_name='users_not_interested')
# lista de usuarios que van a asistir
signed_up = models.ManyToManyField(User,related_name='users_assistants')
"""
=======================================================
```

```
                Servicios de la clase
    ============================================================
    """
    def __str__(self):
        return self.title
    # resto de la clase . . .
```

Al modelo principal se han añadido cinco modelos complementarios que modularizan todos los datos que ofrece un evento:

- *Tag*: añade palabras clave que se relacionan con un evento.
- *Category*: añade categorías a un evento.
- *Photo*: añade fotografías a un evento.
- *Geolocation*: añade la localicación del evento. Este modelo usa una API diferente para poder operar con datos geométricos llamada *django.contrib.gis.db.models*.
- *Comments*: añade comentarios hechos por los usuarios en un evento.

Tanto en el modelo principal como en los complementarios se han añadido funciones de clase para consultas frecuentes o atípicas, dejando las consultas directas ofrecidas por la API implementadas en la vista.

```
class Event(models.Model):
#...
  @classmethod
      # Busca los eventos que contengan una cadena dada en diferentes campos
      def search_string(self,string):
          return Event.objects.filter(
              Q(title__icontains=string) |
              Q(summary__icontains=string))
# De este modo, en la vista se podrá usar esta consulta de manera más intuitiva
queryset = Event.search_string(contenido_a_buscar)
```

En la vista de los eventos se han usado clases predefinidas que ofrece Django para facilitar la creación, la modificación, el listado y el visionado de los detalles (Figura 4.1).

El siguiente fragmento de código define la página de filtrado de eventos, incluyendo la plantilla, el modelo a listar, el paginado y la consulta a la base de datos con las diferentes condiciones:

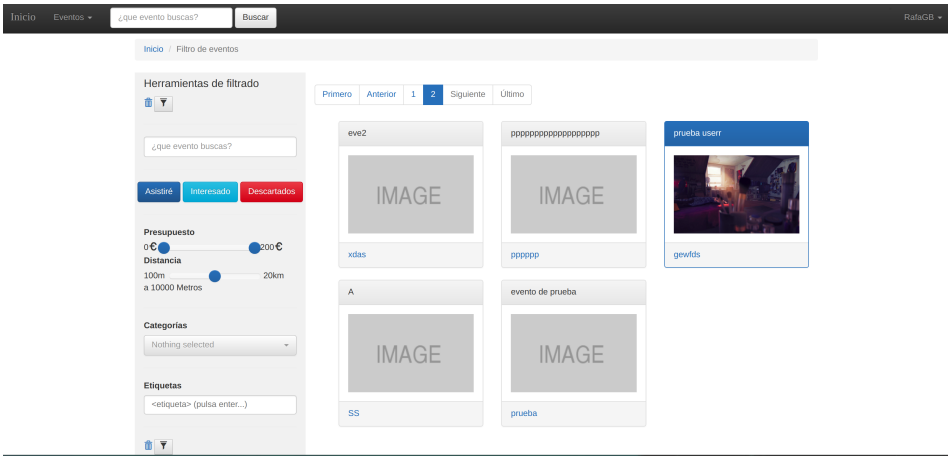

Figura 4.1: Captura de pantalla de la página de filtros

```python
from django.views.generic import ListView

class EventFilterView(ListView):
    model = Event # Nombre del modelo a listar
    context_object_name = 'events' # Nombre en clave
    template_name = 'eventFilter.html' # Plantilla objetivo
    ordering = ['views']
    paginate_by = 9 # Paginacion para la lista incluida
    """
    ------------------------------------------
        Funciones de la clase
    ------------------------------------------
    """
    def get_context_data(self, **kwargs):
        context = super(EventFilterView, self).get_context_data(**kwargs)
         # Lógica para el tratamiento de datos que se mostraran en la plantilla
        # . . .
        return context


    def get_queryset(self):
      # El argumento type se especifica
        # En la llamada a la URL, dentro de la plantilla
        if self.kwargs['type'] == 'search':
            contains = self.request.GET['search']
            queryset = Event.search_string(contains)
        if self.kwargs['type'] == 'own':
            queryset = Event.for_user(self.request.user)
        else:
```

```
        queryset = Event.objects.all()
    # Lógica para el filtrado en la base de datos
      # . . .
      return queryset
```

4.5. Estructura de las plantillas

Las plantillas, en las cuales se muestra el resultado final al usuario, se organizan a partir bloques. El sistema de bloques consiste en definir plantillas modelo donde organizar una estructura general de un archivo *HTML* clásico y extenderla, de tal manera que para todas aquellas plantillas con una estructura en común se comparta la misma base.

Cada bloque debe tener un nombre único que lo identifique. También se permite anidar bloques unos dentro de otros.

Si una plantilla hereda de otra, todo el contenido de la primera debe ir dentro de los bloques predefinidos por la plantilla base. Al llamar a un bloque, reemplazaría todo el contenido que se hubiera añadido dentro del mismo en la plantilla de la cual se extiende. En caso de no hacer uso de alguno de los bloques heredados, su contenido sería el añadido en la base. El orden seguido de los bloques puede variar al extenderse, pero es recomendable seguirlo para garantizar su mantenibilidad.

A continuación se muestra una de las plantillas base del proyecto como muestra de la organización en bloques:

```
{% load staticfiles %}
{% load static %}
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>{% block title %}{% endblock %}</title>
    {% block stylesheet %}
    {% endblock %}
    <script src="{% static 'js/jquery-3.2.1.min.js' %}"></script>
    <script src="{% static 'js/popper.min.js' %}"></script>
    {% block javascript %}
    {% endblock %}
  </head>
  <body>
    {% block body %}
      {% block header %}
```

```
      {% endblock %}
      {% block base_container %}
      <div class="container">
        <ol class="breadcrumb my-4">
          {% block breadcrumb %}
          {% endblock %}
        </ol>
        {% block content %}
        {% endblock %}
      </div>
      {% endblock %}
    {% endblock body %}
    {% block footer %}
    {% endblock %}
  </body>
  <script src="{% static 'js/bootstrap.min.js' %}"></script>
  {% block javascriptBottom %}
  {% endblock %}
</html>
```

Como ejemplo de plantilla extendida que haga uso de la base anterior se muestra el siguiente ejemplo:

```
{% extends 'base.html' %}

{% block title %}titulo de la plantilla{% endblock %}

{% block stylesheet %}
  <!-- nuevos archivos css -->
{% endblock %}

{% block javascript %}
    <!-- nuevos archivos js -->
{% endblock %}

{% block content %}
  <!-- el contenido que se desee -->
{% endblock %}
```

Capítulo 5

Los usuarios y su relación con los eventos

En este capítulo se detallarán las interacciones implementadas entre las dos aplicaciones creadas para el proyecto: la aplicación que gestiona los usuarios (*accounts*) y la que gestiona los eventos (*events*).

5.1. Interacciones directas

Un evento debe ser creado por un usuario. Para ello se ha implementado un formulario propio de creación de eventos y uno de edición basado en la vista de Django `UpdateView`.

5.1.1. Creación de eventos

El formulario de creación (figura 5.1) abarca el modelo `Event` con la información básica y los modelos `Tag`, `Category`, `Geolocation` y `Photo` con la información añadida.

El siguiente fragmento de código detalla la implementación de la clase encargada de crear y validar el formulario del modelo `Event`:

```
class EventForm(forms.ModelForm):
    description = forms.CharField(
      # Configuración personalizada
    )

    class Meta:
```

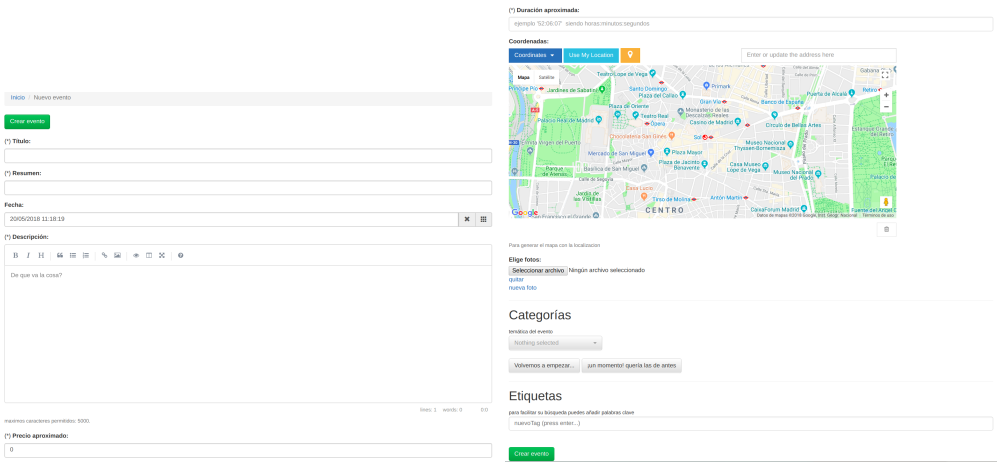

Figura 5.1: Capturas de pantalla del formulario de creación de un evento

```
  model = Event
# Elementos a mostrar del modelo
  fields = [
   'title',
   'summary',
      'date',
      'description',
   'budget',
   'duration'
  ]
  # Cambia el nombre del campo por el que se desea mostrar en HTML5
  labels = {
      'date': _('Fecha'),
      'title': _('Titulo'),
      'summary': _('Resumen'),
      'budget': _('Precio aproximado'),
      'duration': _('Duracion aproximada')
  }
# Cambia la apariencia del input por defecto
  widgets = {
      'date': DateTimeWidget(attrs={'id':"id_date"}, usel10n = True,
         bootstrap_version=3),
      'duration': forms.TextInput(attrs={'placeholder': "ejemplo '52:06:07'
         siendo horas:minutos:segundos"})

  }
```

El siguiente fragmento de código muestra la vista usada para la creación de un evento:

```
@login_required
def NewEvent(request):
    # Función requerida para el manejo de múltiples fotos
    PhotoFormSet = modelformset_factory(model=Photo,form=PhotoForm, extra=1)

    if request.method == 'POST':
        form = EventForm(request.POST)
        formGeo = GeolocationForm(request.POST)
        formset = PhotoFormSet(request.POST or None, request.FILES or None)

        dateform = request.POST["date"]

        if all([form.is_valid(),formset.is_valid(),formGeo.is_valid()]):
         # Tratamiento de Geolocation
            myGeo = formGeo.save()

            # Tratamiento de Event
            myGeo = formGeo.save()
            event = form.save(commit=False)
            event.geopos_at = myGeo
            event.created_by = request.user
            event.save()

            # Tratamiento de Photo
            for tmp_form in formset.cleaned_data:
                # . . .

            # Tratamiento de Tag

            myTags = request.POST['myTags']
            arrayTags = myTags.split(',')

            for tag in arrayTags:
                # . . .

            # Tratamiento de las Category

            arrayCategories = request.POST.getlist('myCategories')
            for category in arrayCategories:
                # . . .

            # Redirección a la pagina de detalles al concluir la creación
            return redirect('eventDetails', pk=event.pk)
    else:
```

```
        # Inicialización de formularios

    return render(
            # Datos para la plantilla
        )
```

5.1.2. Modificación de eventos

El formulario de edición (figura 5.2) muestra una estructura idéntica al formulario de creación descrito en la sección anterior con los campos inicializados usando la información del evento a modificar.

Los modelos `Tag`, `Category` y `Photo` requieren una estructura de listado. Su modificación puede implicar la eliminación de elementos, por lo que esta estructura queda contemplada en la lógica de la vista. Para su implementación, como se comenta en la descripción de la sección, se ha utilizado la vista `UpdateView` de Django, ya que facilita la inicialización de los elementos del modelo para los múltiples formularios.

A continuación se detalla la implementación de la vista (los formularios utilizados son los mismos que para la creación del evento):

```
@method_decorator(login_required, name='dispatch')
# Controla que un usuario solo pueda modificar sus propios eventos
@method_decorator(user_is_event_author, name='dispatch')
class EventUpdateView(UpdateView):
    model = Event
    template_name = 'update_event.html'
    context_object_name = 'event'
    form_class = EventForm
    formGeo_class = GeolocationForm
    """
    ------------------------------------------------------------
        Funciones de la clase
    ------------------------------------------------------------
    """
    def get_context_data(self, **kwargs):
        # Recuperamos argumentos ya inicializados
        context = super(EventUpdateView, self).get_context_data(**kwargs)

        if not 'errors' in context:
            context['form'] = self.form_class(instance=self.object)
            # Se incorpora al contexto todos los formularios inicializados
```

```
        return context

    def get(self, request, *args, **kwargs):
        super(EventUpdateView, self).get(request, *args, **kwargs)
        form = self.form_class
        # Resto de formularios
        return self.render_to_response(self.get_context_data(
            object=self.object, form=form ,formGeo=formGeo ,formset=formset))

    def post(self, request, **kwargs):
        self.object = self.get_object()
        form = self.form_class(request.POST)
        formGeo = self.formGeo_class(request.POST)
        PhotoFormSet = modelformset_factory(model=Photo, form=PhotoForm,
            formset=BasePhotoFormSet, can_delete=False)
        formset = PhotoFormSet(request.POST or None, request.FILES or None)
        if all([form.is_valid(),formGeo.is_valid(),formset.is_valid()]):
            """
                            Tratamiento de Event
            ..................................................

            """
            self.object.title = form.cleaned_data['title']
            # . . .
            """
                            Tratamiento de Geolocation
            ..................................................

            """
            updateGeo = Geolocation.objects.get( pk=self.object.geopos_at.pk )
            updateGeo.coordinates = formGeo.cleaned_data['coordinates']
            updateGeo.save()
            # Una vez guardado se guarda el evento
            self.object.save()
            """
                            Tratamiento de Tags
            ..................................................

            """
            primalTags = list( Tag.objects.filter( events_tags=self.object
                ).values_list( 'name_tag', flat=True ) )
            myTags = request.POST["myTags"]
            arrayTags = myTags.split(',')
            for tag in arrayTags:
                newTag = None
```

```
                tagExist = Tag.objects.filter(name_tag=tag).exists()
                if not tagExist:
                    # Crear nuevo tag
                elif tag not in primalTags:
                    # Se agrega a la lista del evento
                else:
                    # Se descarta ya que no hay cambios
            # Los tags no descartados han sido eliminados y se borran de la lista
            for tag in primalTags:
                removeTag = Tag.objects.get(name_tag=tag)
                if removeTag.events_tags.count() == 1:
                    # Este tag solo se usaba na vez y queda borrado
                else:
                    # Este tag existe en otros eventos por lo que
                    # Solo se elimina de este evento

    # Resto de modelos con una estructura de lista
      else:
          return self.render_to_response(
            self.get_context_data( errors=True, form=form, formGeo=formGeo,
               formset=formset ))
```

5.2. Intereses y asistencia a eventos

Para un evento, todo usuario (menos su creador) puede elegir entre tres opciones (figura 5.3) que marcan su relación con el evento, siendo excluyentes entre sí:

- **Confirmar asistencia**: Indica su intención de ir al evento.
- **Estar interesado**: Indica un interés por el evento, aunque no la asistencia definitiva al mismo. Puede utilizarse a modo de marcador por si desea asistir más adelante.
- **No estar interesado**: Indica un rechazo al evento evitando que vuelva a salir en el buscador y no molestar con esta información al usuario. Esta opción se puede deshacer filtrando por los eventos no deseados y desmarcando la casilla.

Al hacer clic en una de estas opciones se mandará una petición AJAX para actualizar en la base de datos la preferencia del usuario identificado, por lo que no será necesario recargar la página. El botón seleccionado cambiará en función de su estado anterior y se comprobará si otra opción había sido seleccionada antes para modificarla. Para ello se hace uso de AJAX y Javascript que permite cambios dinámicos en la página actualmente cargada en el navegador.

Fragmento de código que envía la petición AJAX:

```
<script>
// . . .
function ajax_vote(tipo){
      $.ajax({
        type: "POST",
        url: '/eventFlowControl/'+tipo+'/',
        data: {
          'event_pk': '{{object.pk}}',
          'csrfmiddlewaretoken' : '{{ csrf_token }}'
        },
        dataType: 'json',
        success: function (data) {
        //tratamiento de la respuesta del servidor
        //en caso de no haber habido errores
        }
      });
    }
</script>
```

A continuación se muestra fragmento de código que procesa la petición AJAX y devuelve una respuesta al usuario:

```
@login_required
def EventFlowControl(request,**kwargs):
    if request.method == 'POST':
        id_event = request.POST['event_pk']
        user = request.user
        event = Event.objects.get(pk=id_event)

        option = None
        tipo = kwargs['type']
        remove_add = None
        if tipo == "interested":
            # . . .
        elif tipo == "attendant":
            # . . .
        elif tipo == "not_interested":
            # . . .
        else:
            # . . .
        data = {
            'remove_add' : remove_add
        }
    else:
        data = {}
    return JsonResponse(data)
```

5.3. Comentarios

Si el usuario se encuentra identificado en la página de cada evento se mostrará la opción de añadir un comentario mediante un editor de texto y una lista de todos los comentarios ya publicados ordenados por fecha de publicación (figura 5.4).

Al pulsar la opción de publicar se enviará una petición AJAX al servidor para guardar en la base de datos la información pertinente al comentario. Al mismo tiempo, sin recargar la página, se mostrará con una rápida animación en la parte superior de la lista de comentarios el comentario que se acaba de publicar y un mensaje de éxito.

La aplicación permite a un usuario borrar cualquier comentario que este haya publicado previamente. Se hará uso de la misma petición AJAX con diferentes parámetros de entrada y borrará a continuación de la base de datos dicho comentario.

Fragmento de código que envía la petición AJAX.

```
<script>
  function ajax_comment(tipo,messageID,commentID){
    if(tipo =="remove" && !confirm("Estas seguro que deseas
       borrar el comentario?")){
        return;
    }
      $.ajax({
        type: "POST",
        url: '/eventCommentsControl/'+tipo+'/',
        data: {
          'event_pk': '{{object.pk}}',
          'pk_comment': commentID,
          'message': simplemde.value(),
          'csrfmiddlewaretoken': '{{ csrf_token }}'
        },
        dataType: 'json',
        success: function (data) {
        //tratamiento de la respuesta del servidor
        //en caso de no haber habido errores
        }
      });
    }
</script>
```

Fragmento de código que procesa la petición AJAX y devuelve una respuesta al usuario:

```
@login_required
def EventCommentsontrol(request,**kwargs):
    if request.method == 'POST':
        id_event = request.POST['event_pk']
        message = request.POST['message']
        user = request.user
        newComment = Comments()
        if Event.objects.filter(pk=id_event).exists():
            event = Event.objects.get(pk=id_event)
            option = None
            tipo = kwargs['type']
            remove_add = None
            if tipo == "create":
                # . . .
            elif tipo == "remove":
                # . . .
            else:
                feedback='error'
```

```
            data = {
                'feedback' : feedback,
                'message' : message,
                'time' : ' ahora mismo',
                'type' : tipo,
                'comment_pk': newComment.pk
            }
        else:
            print("evento NO existe")
            data = {}
    else:
        data = {}
    return JsonResponse(data)
```

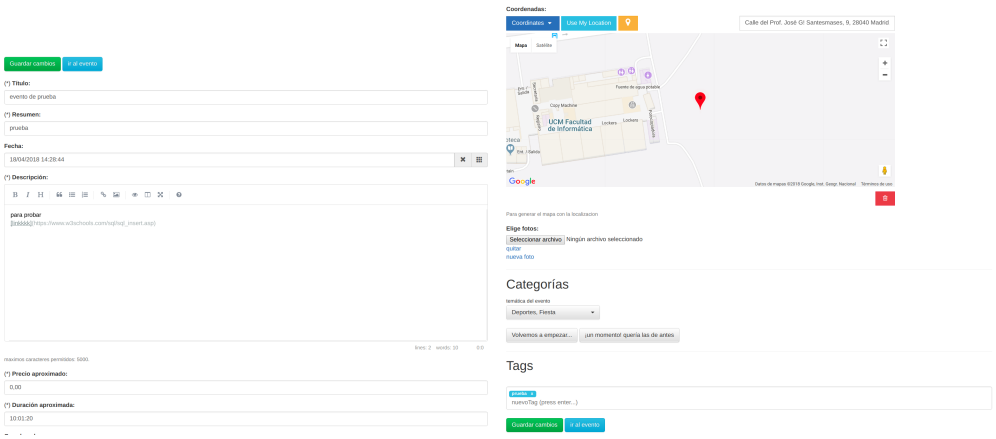

Figura 5.2: Capturas de pantalla del formulario de edición de un evento

Figura 5.3: Captura de pantalla de los botones antes y después de seleccionar una opción

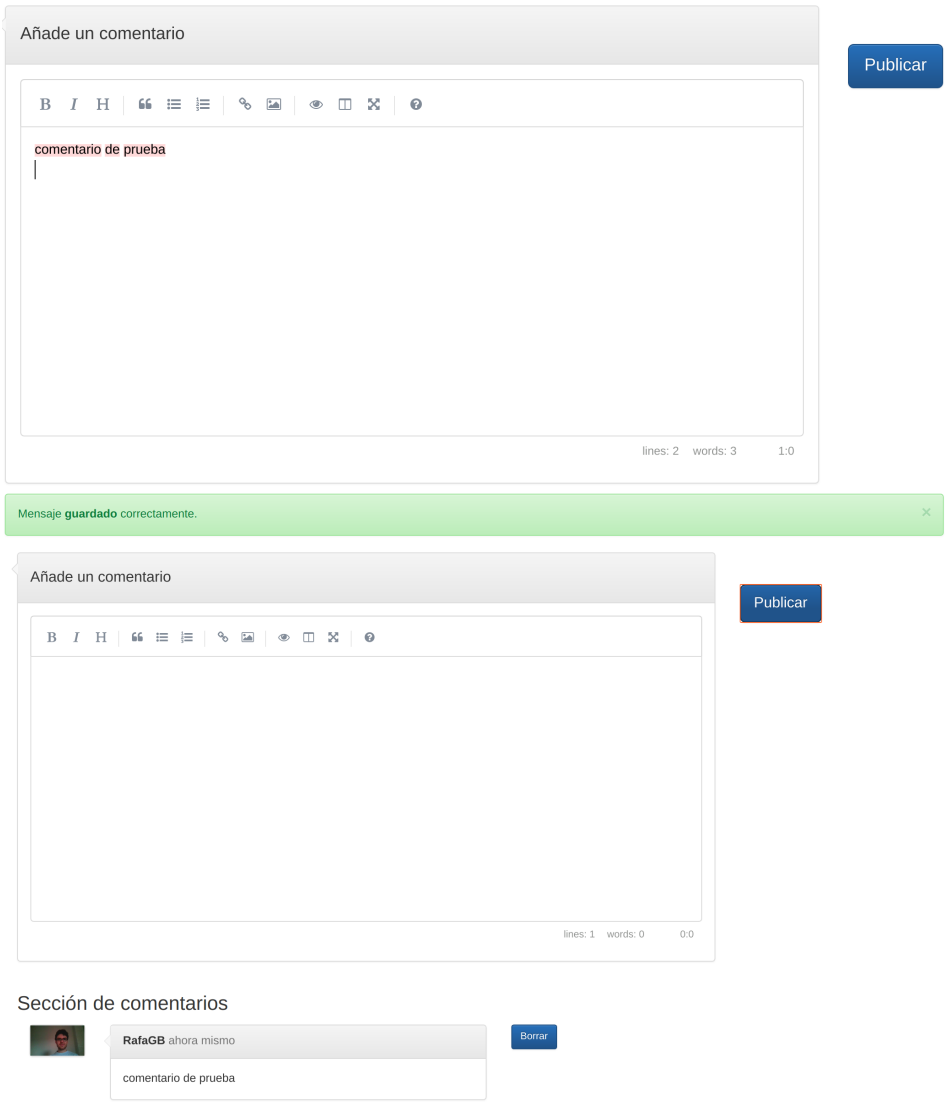

Figura 5.4: Captura de pantalla de un comentario antes y después de ser publicado

Capítulo 6

Búsqueda avanzada de eventos

En este capítulo se contará en profundidad la herramienta de búsqueda y filtrado de eventos implementada para la web (Figura 6.1).

6.1. Realización de la consulta

Para consultar los eventos usando filtros diferentes se hace uso de la API que ofrece Django mediante las *queryset*. Una *queryset* es una abstracción de una consulta sobre la BD. Puede construirse directamente a partir de una tabla, estableciendo condiciones de búsqueda sobre la misma, o bien a partir de otro *queryset*, estableciendo condiciones de búsqueda adicionales a los ya descritos por el mismo. Esto es ideal para componer diferentes filtros dependientes entre sí de manera modular.

Para su implementación en el proyecto, en primer lugar se vuelca en la *queryset* una consulta más genérica en función del parámetro de entrada `type` definido en la URL. Este parámetro de entrada decidirá si se consulta sobre los propios eventos del usuario autenticado o si se hace sobre todos los eventos de la base de datos.

Una vez se ha definido el *queryset*, se comprueban las variables de tipo `GET` en el `request` recibidas desde el navegador y se tratan para establecer condiciones adicionales sobre lo ya filtrado hasta obtener la consulta que el usuario ha solicitado.

A continuación se detallará cada filtro y su implementación definida en la aplicación *events/views.py* en la clase `EventFilterView`:

```
class EventFilterView(ListView):
# Resto de la clase
```

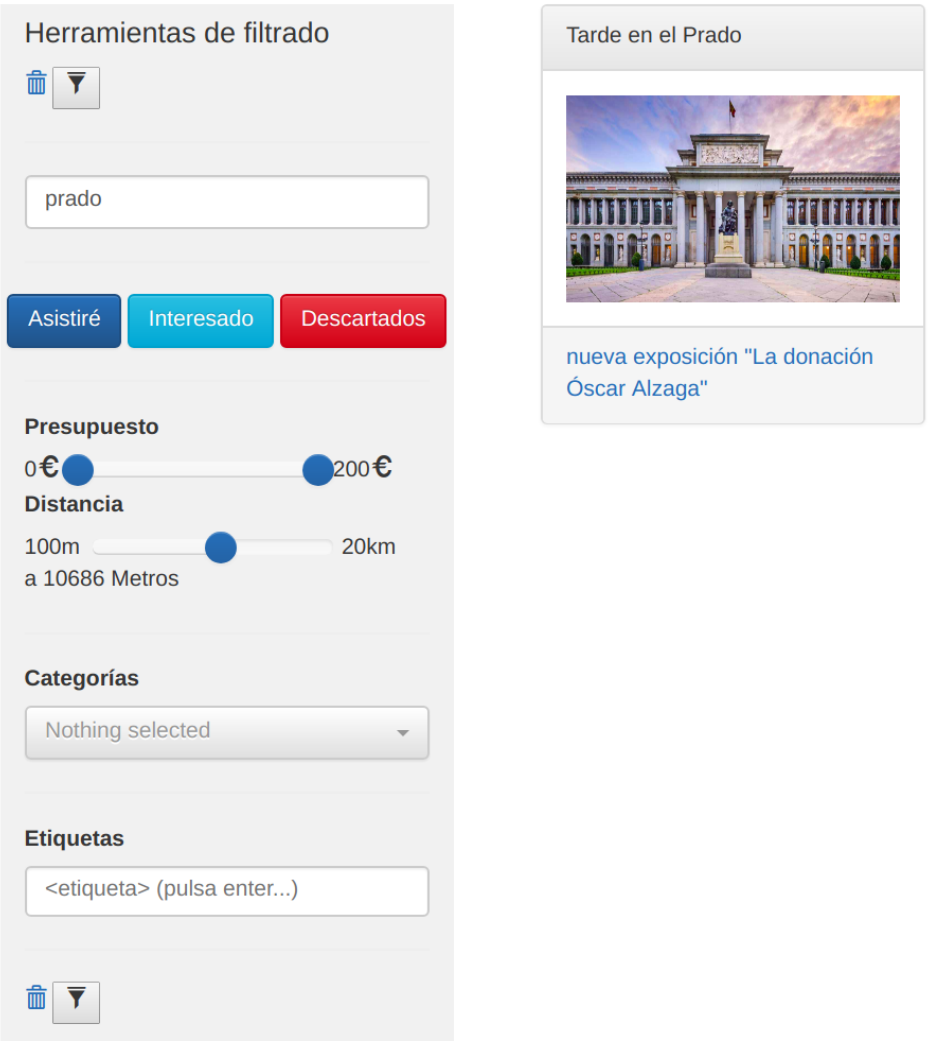

Figura 6.1: Captura de pantalla con un ejemplo de uso del filtrado

```
def get_queryset(self):

        """
        ----------------------------------------------------------
            Se define la queryset
        ----------------------------------------------------------
        """
        if self.kwargs['type'] == 'own':
         # Solo eventos del usuario indicado
            queryset = Event.for_user(self.request.user)
        else:
         # todos los eventos
            queryset = Event.objects.all()


        try:
```

El filtro `search` es un filtro de texto para los valores que más caracterizan un evento: su título y su resumen.

```
            if('search' in self.request.GET and self.request.GET['search']):
                contains = self.request.GET['search']
                queryset = (queryset & Event.search_string(contains))
```

El filtro `distance` se encarga de mostrar los eventos más cercanos a la posición actual del usuario en función de un valor entero que indica la distancia en metros sobre la que se quiere acotar. La posición del usuario viene definida por `lat` y `lng`, variables enviadas desde el navegador cuando el usuario permite que el navegador detecte su localización. En caso contrario, no se llevará a cabo el filtro.

```
            if('lat' in self.request.GET and self.request.GET['lat'] \
                and 'lng' in self.request.GET and self.request.GET['lng'] \
                and 'distance' in self.request.GET and
                    self.request.GET['distance']):
                distance = self.request.GET['distance']
                distance = int(distance)
                # En el servidor se comprueba que la distancia sea
                # como maximo 20 kilometros
                distance = distance if distance <= 200000 else 200000
                lat , lng =
                    (float(self.request.GET['lat']),float(self.request.GET['lng']))
                ref_location = Point(lat, lng)
                queryset = (queryset & Event.distance_range(ref_location,distance))
            """
```

```
Función definida en el modelo
Busca los eventos que se encuentren en un rango
de 'n' metros de la posicion actual
---
@classmethod
def distance_range(self,location,meters):
return Event.objects.filter( geopos_at__coordinates__distance_lt=(
    location, D( m=meters )) )
"""
```

El filtro `tags` recibe una lista de palabras clave sobre la que se itera y se filtra para cada una de ellas. Para el filtro `categories` se una implementación similar, recorriendo cada una de las categorías recibidas.

```
if 'tags' in self.request.GET and self.request.GET['tags']:
    tags = self.request.GET['tags'].split(',')
    for tag in tags:
        queryset = queryset.filter(tags__name_tag=tag)

if 'categories' in self.request.GET and
    self.request.GET['categories']:
    categories = self.request.GET.getlist('categories')
    for category in categories:
        queryset = queryset.filter(categories__name_category=category)
```

El filtro `budget` acota el precio estimado de los eventos que debe devolver la consulta, a partir de un precio máximo y un precio mínimo.

```
if 'budget' in self.request.GET and self.request.GET['budget']:
    budget = self.request.GET['budget'].split(',')
    queryset = (queryset & Event.range_prices(budget[0],budget[1]))
```

Si el usuario está autenticado, podrá acceder al filtro de `interestGroup` donde se seleccionan aquellos eventos en los que se ha especificado que se asistirá, que se está interesado o que no se está interesado:

```
if self.request.user.is_authenticated():
    if 'interestGroup' in self.request.GET and
        self.request.GET['grupoIntereses']:
        interestGroup = self.request.GET['grupoIntereses']
        if grupoIntereses =="attendant":
            queryset = ( queryset &
                Event.objects.filter(signed_up__id=self.request.user.pk)
```

```
                )
            elif interestGroup =="interested":
                queryset = ( queryset & Event.objects.filter(
                    interested_in__id=self.request.user.pk ) )
            elif interestGroup =="removed":
                queryset = ( queryset & Event.objects.filter(
                    not_interested_in__id=self.request.user.pk ) )
            else:
                queryset = queryset.exclude(
                    not_interested_in__id=self.request.user.pk )
        else:
            queryset = queryset.exclude(
                not_interested_in__id=self.request.user.pk )
```

Finalmente se ordenan los eventos por número de visitas recibidas. Si durante cualquier etapa de filtrado ha ocurrido un error, se responderá con una consulta de todos los eventos.

```
            queryset = queryset.order_by('views')
        except:
            # Genero un mensaje de error interno en un log
            queryset = Event.objects.all()
        return queryset
```

6.2. Tratamiento de la lista en la plantilla

Tras haber tratado la consulta en profundidad, la plantilla `eventFilter.html` recibe una lista con los eventos que finalmente han quedado tras los distintos filtros y los muestra en una cuadrícula de tres filas y tres columnas, utilizando paginación en caso necesario.

El siguiente fragmento de código de la plantilla muestra cómo es tratada la lista de eventos:

```
<div class="col-sm-9">
{% include 'includes/pagination.html' %}
{% if events %}

<ul>
  <!-- contador para controlar la extension de la fila -->
  {% assign count 0 %}
  {% for event in events %}
    {% ifequal count 0 %}
    <div class="row">
```

```
    {% endifequal %}
      <a href="/{% if view.kwargs.type == 'own' %}updateEvent{%
         else %}event{% endif %}/{{ event.pk }}">

      <div class="col-sm-4">
        <div class="
        {% if user in event.interested_in.all %}
        panel panel-info
        {% elif user in event.signed_up.all %}
        panel panel-primary
        {% elif user in event.not_interested_in.all %}
        panel panel-danger
        {% else %}
        panel panel-default
        {% endif %}
        ">
          <div class="panel-heading">{{ event.title }}</div>

          <div class="panel-body"><img src="
          {% if event.photo_set.all %}
          {{event.photo_set.all.0.picture.url}}
          {% else %}
          https://placehold.it/200x130?text=IMAGE
          {% endif %}
          " class="img-responsive" style="width:100%;height: 130
             px" alt="Image"></div>
          <div class="panel-footer">{{event.summary}}</div>
        </div>
      </div>
      </a>
    {% ifequal count 2 %}
    </div>
      <!-- reseteamos contador -->
      {% assign count 0 %}
    {% else %}
      <!-- count++ -->
      {% increment count %}
    {% endifequal %}

  {% endfor %}
  <br>
</ul>
{% else %}
<p>No se han encontrado coincidencias con su busqueda</p>
{% endif %}
```

```
{% include 'includes/pagination.html' %}

</div>
```

Capítulo 7

Recomendación de eventos

En este capítulo se detallará la funcionalidad de recomendación de eventos implementada para usuarios autenticados. Las recomendaciones se muestran en función de los eventos a los que los usuarios eligieron asistir.

7.1. Estructura básica

Para llevar a cabo un sistema de recomendaciones se necesita de información relevante que represente los gustos de cada usuario. Para ello se ha creado el modelo *recommender*, el cual contiene una lista de eventos considerados relevantes para un determinado usuario.

A continuación se muestra el modelo del recomendador:

```
class MyRecommender(models.Model):
    user = models.IntegerField(primary_key=True)
    id_events = ArrayField(models.IntegerField(blank=True),default=list,
        null=True)

    def __str__(self):
        return str(self.user)
```

Para poder obtener la lista de eventos recomendados de cada usuario se ha tenido en cuenta los eventos en los que seleccionaron 'asistiré' para, con ello, averiguar qué usuarios eligieron de la misma manera. Una vez hallados los usuarios con las mismas elecciones, se buscan aquellos eventos a los que estos últimos van a asistir, pero el usuario al cual se está recomendando aún no lo ha hecho, y agruparlos en una lista sin repeticiones.

Figura 7.1: Captura de pantalla de recomendaciones para un usuario

7.2. Algoritmo utilizado

Para hallar la lista de recomendaciones se ha creado un archivo de tipo `.CSV` con tres columnas: clave primaria de eventos (evento con asistentes), clave primaria de usuarios (usuario que asiste) y visitas del eventos.

A continuación se muestra el código con la instrucción a la base de datos para crear el archivo:

```
from django.db import connection
# e parametro de entrada path indica en que ruta se desea guardar
def restart_csv(path):
    with connection.cursor() as cursor:
        cursor.execute("copy (SELECT ev.id,signedup.user_id,ev.views
        FROM \"public\".\"events_event\" AS ev
        LEFT JOIN \"public\".\"events_event_signed_up\" AS signedup
        ON ev.id=signedup.event_id
        ORDER BY -ev.views) TO '"+path+"' DELIMITER ',' CSV")
```

En dicho archivo se guarda toda la información sobre a asistencia a los eventos de todos los usuarios de la base de datos. De este modo puede manejarse a modo de matriz a gran escala.

Para poder tratar los datos en un tiempo razonable con la matriz generada, se crea una submatriz a partir del *dataframe* entero usando la librería `cross_validation` de `sklearn` de Python, que recogería un porcentaje inferior al total de las filas del archivo previamente volcado en el *dataframe* utilizando la librería `pandas`.

```
import pandas as pd

header = ['event_id', 'user_signedUp_id', 'views']
df = pd.read_csv(self.csv_path, sep=',', names=header)
```

A continuación se muestra el fragmento de código donde queda implementada la lógica descrita en la sección 7.1:

```
class Recomender(CronJobBase):
# . . .
# Diccionario con el resultado final de todas las listas
# asociando una a cada usuario
dictRecommenders = {}
       for x in unique_user_ids:
           dictRecommenders[x]=[]
"""
Se trata cada evento individualmente con la matriz para hallar
eventos relevantes en cualquier usuario
"""
for event_id in unique_event_ids:
           event_pos = event_id_pos[event_id]
           row = train_data_matrix[event_id_pos[event_id],:]
           pos_similar_users = []
           recommended_events=[]
           # Se buscan coincidencias entre usuarios mirando cada fila
           for index,elem in enumerate(row):
               if int(elem)>0:
                   pos_similar_users.append(index)
                   recommended_events.append(int(elem))
           if len(pos_similar_users)>1:
               for index,(posUser,idEvent) in
                   enumerate(zip(pos_similar_users,recommended_events)):
                   pos_other_similar_users = pos_similar_users
                   pos_other_similar_users.remove(posUser)
                   # Para cada usuario con alguna coincidencia
                   for j in pos_other_similar_users:
                       # Se comprueban eventos recomendables
                       # en la columna de la matriz
                       col = train_data_matrix[:,j]
                       for y in col:
                           if y>0 and y != event_id and y not in
                               dictRecommenders[unique_user_ids[posUser]]:
                              dictRecommenders[unique_user_ids[posUser]].append(int(y))
```

Una vez conseguido el diccionario con todas las listas de recomendaciones se vuelcan en la base de datos:

```
for key, value in .items():
  pos = user_signedUp_id_pos[key]
  col = train_data_matrix[:,pos]
  for i in col:
  if int(i) in value:
   value.remove(i)
  tmpRecommender = MyRecommender(user=int(key),id_events=value)
tmpRecommender.save()
```

Este proceso se repetirá automáticamente de manera periódica para ir actualizando las recomendaciones de los usuarios. Dichas recomendaciones se mostrarán en la página principal de la web como se muestra en la figura 7.1. En caso de no existir ninguna recomendación se mostrará un mensaje informativo (Figura 7.2).

Figura 7.2: Captura de pantalla de recomendaciones para un usuario

Capítulo 8

Conclusiones y trabajo futuro

En este capítulo se discutirán hasta qué punto se han llevado a cabo los objetivos propuestos por el trabajo, las dificultades que han supuesto y qué resultaría interesante incorporar.

8.1. Objetivos alcanzados

La idea básica del trabajo de llevar a cabo una página web funcional sobre la gestión de eventos se ha llevado a cabo superando las expectativas, las cuales se veían limitadas por el aprendizaje de una nueva tecnología y el tiempo de adaptación que supone. Tras el aprendizaje de *Django* y al ver su potencial se fueron añadiendo funcionalidades que no habían sido pensadas en un principio y que aportan valor a la web (por ejemplo, comentarios o palabras clave en los eventos), y otras muchas que no se han llevado a cabo por limitaciones de tiempo y que serían de utilidad (por ejemplo, un calendario donde gestionar todas las actividades).

El diseño de la interfaz trajo dificultades al añadir a los formularios diseños de terceros que facilitaban el introducir una dirección en el mapa o añadir una fecha al evento a través de un calendario interactivo. Esto requería aprender nueva documentación sobre su uso, pero la mayor dificultad fue aprender el funcionamiento de los `formset` en *Django*. Un `formset` permite disponer de formularios dinámicos, en tanto que su número de componentes puede variar a medida que el usuario introduce los datos. En particular, el `formset` ha sido utilzado para gestionar múltiples fotografías en un mismo formulario. El uso de los `formset` no disponía de mucha documentación y supuso un parón en el desarrollo de dicho formulario optando por continuar con el resto de funcionalidades hasta adquirir más experiencia con la tecnología.

La elaboración del servidor, una vez dominado el *framework*, ha llevado a un proyecto organizado, sin graves problemas de funcionamiento y con una escalabilidad potencial, siendo la parte más elaborada la parte de gestión de la base de datos. Encontré diversos problemas a la hora de filtrar por distancia al usuario un evento, ya que toda la documentación que encontré sobre el tratamiento de datos geométricos con *Django ORM* estaba enfocada al uso de un tipo de datos concreto dentro de un sistema gestor de bases de datos concreto (*PostgreSQL*). El usado hasta el momento (*SQLite*) utilizaba una representación de datos distinta, por lo que se adaptó el mecanismo de almacenamiento al estándar que requería la herramienta.

Para concluir, el último objetivo era implementar un sistema de recomendaciones coherente con cada usuario. Tras indagar en el funcionamiento de las librerías de *machine learning* y dado que el tipo de información que se guarda en la base de datos es suficientemente esclarecedor para hallar los gustos de cada usuario registrado de una manera más directa, se tomó la decisión de usar un algoritmo propio para llevar a cabo dichas recomendaciones. Este algoritmo se podría mejorar si se tienen en cuenta otros factores como las categorías u otros atributos que se añadieran en un futuro a la web.

8.2. Trabajo futuro

La página web admite múltiples añadidos. A continuación se enumerarán las siguientes propuestas:

- **Página de calendario**

  Para facilitar la organización al usuario, mejoraría mucho la aplicación tras la incorporación de una interfaz con un calendario donde poder ver los eventos a los que va a asistir con un acceso directo a cada uno de ellos. En dicho calendario los eventos dispondrían de opciones rápidas de cancelación o cambio de opinión.

- **Creación de cuenta a través de terceros**

  Añadir la opción de crear cuenta usando la API de *Facebook* y *Google* de tal manera que, con un solo click, el usuario tendría acceso a todas las funcionalidades más cómodamente sin tener que crear una cuenta en la aplicación web..

- **Incorporación de redes sociales**

  Añadir la opción de compartir los eventos en las redes a través de diferentes APIs como *Twitter* o *Facebook* daría mucha más visibilidad a la web.

- **Sistema de puntuación**

  Incorporar la posibilidad de votar un evento daría una información muy valiosa para mejorar las recomendaciones personalizadas. También se usaría para añadir una opción de filtrado más.

- **Subida del proyecto a un *Docker***

  Subir a una platafoma de despliegue (como *Docker*) la web para que el proyecto pueda ser usado por cualquier persona. Una vez se haya subido cambiar la forma de almacenar el contenido multimedia (en este caso, las fotografías de cada evento y el avatar de cada usuario) en un sistema de almacenamiento en la nube como *Amazon S3* usando un *web service*.

- **Mejora en el sistema de recomendaciones**

  Por falta de tiempo, no se ha ahondado en la funcionalidad de recomendaciones que dispone la pagina web, por lo que sería interesante investigar nuevas alternativas al algoritmo ya implementado o añadir nuevos condicionantes con la información que ya se dispone de cada usuario como sus categorías preferidas.

- **Crear una versión para móvil**

  Llevar la aplicación a una versión móvil multi-plataforma adaptando la interfaz web a dispositivos móviles daría mucho más valor al proyecto.

Capítulo 9

Conclusions and future work

In this chapter we will discuss to what extent the objectives introduced in this work have been carried out, the difficulties they have entailed and what it would be interesting to incorporate.

9.1. Achieved goals

The main goal of developing a web page for event management has been carried out successfully. The initial expectations, which were limited due to the necessity of learning a new technology (with the amount of time it involves) have been surpassed. After I learnt and discovered the potential of *Django*, I was able to incorporate a few features that had not been planned in advance (such as comments and keywords in events), and identify several other features that, although they would be useful, have not been implemented due to time constraints (for example, a calendar to display and manage all the activities).

The design of the interface involved some issues when integrating third-party components into forms. It turns out that these components make the task of entering an address (via a map) or selecting a given date (through an interactive calendar) easier. but the addition of these components required studying further documentation. Nevertheless, the biggest issue was found when dealing with Django's *formset*. A *formset* allows a web page to dynamically change the number of form components. This has been used in the form that manages the pictures attached to an event. The scarcity of suitable formset's documentation implied a break in the development of this functionality until I had acquired more experience with this technology.

Once the learning curve of the *framework* has been overcome, the development of the server's functionality has led to an well-structured project, without serious operational issues

and with a potential scalability, being the database management layer the most elaborated part. I run into several issues when trying to filter the event search results by distance, since all the documentation available about processing geographical data with *Django ORM* was specific to a given data type in the context of a given relational database management system (*PostgreSQL*). The RDBMS that had been used so far (*SQLite*) used a different representation, so the storage mechanism had to be adapted to the standard representation required by *Django ORM*.

To conclude, the last goal was to implement a recommender system adapted to each user's needs. After surveying the various *machine learning* libraries, and given that the type of information stored in the database is simple enough to infer recommendations for a given user in a more direct way, I decided to use a specific algorithm to carry out this phase. However, this algorithm could be improved if other factors are taken into account, such as categories or other attributes associated with an event.

9.2. Future work

The website supports multiple additions. The following proposals will be listed below:

- **Calendar view**

  In order to make the user organization easier, the inclusion of a calendar would greatly improve the web application. In this calendar the user would be able to find the events that in which he is expected to attend, with a direct access to its information. This calendar would also provide shortcuts for changing the user's preference (for example, cancelling the attendance).

- **Creating an account through third-party authentication systems**

  It would be useful to allow the user to create an account by using the API of *Facebook* and *Google* in such a way that with a single click the user would have access to all the functionalities more comfortably without having to create an account in the web application.

- **Incorporation of social networks**

  Adding the option to share events on networks through different APIs such as *Twitter* or *Facebook* would give much more visibility to the web.

- **Punctuation system**

  Incorporating the possibility of voting an event would give a very valuable information to improve the personalized recommendations. It would also be used to add one more filtering option.

- **Upload the project to a *Docker***

  It is feasible to upload the web application to a deployment platform (like *Docker*), so that the project can be used by anyone. Once it has been uploaded, the way of storing the multimedia content can be changed, for instance by storing the pictures of each event and each user's avatar in a cloud storage system such as *Amazon S3*, accessed through a *web service*.

- **Improvement in the recommendations system**

  Due to the lack of time, I have not had the chance to deepen into the recommender system, so it would be interesting to research new alternatives to the algorithm already implemented, or include new constraints from the information that is already available in the database, such as the categories that an user prefers.

- **Create a mobile version**

  The development of a multi-platform mobile version of the application, by adapting the web interface to mobile devices, would give much more value to the project.

Bibliografía

[1] Django Software Foundation. Django making queries. `https://docs.djangoproject.com/en/2.0/topics/db/queries/`, 2017. [Online; accedido 19-Diciembre-2018].

[2] Vitor Freitas. simple is better than complex. `https://simpleisbetterthancomplex.com/series/2017/09/04/a-complete-beginners-guide-to-django-part-1.html`, 2017. [Online; accedido 16-Noviembre-2017].

[3] Nigel George. *Mastering Django: Core, the complete guide of Django 1.8 LTS*, volume 1 of *2*. GNW, 2016.

[4] Stefan KottWitz. *Latex cookbook*, volume 1 of *1*. PACKT publishing, 2015.

[5] Luciano Ramalho. *FLuent Python*, volume 3 of *3*. O'Reilly, 2015.

[6] Jake Spurlock and Dave Winer. *Mastering Django: Core, the complete guide of Django 1.8 LTS*, volume 1 of *1*. O'Reilly, 2013.

[7] Google Maps team. Google Maps Platform Documentation. `https://developers.google.com/maps/documentation/`, 2017. [Online; accedido 04-Diciembre-2017].

[8] PostgreSQL Tutorial team. PostgreSQL Tutorial. `http://www.postgresqltutorial.com/`, 2018. [Online; accedido 09-Marzo-2018].